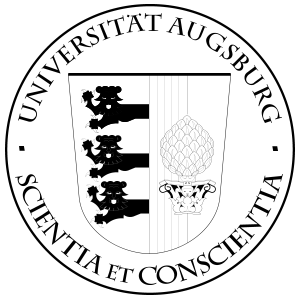
Universität Augsburg



Embedding Rely-Guarantee Reasoning in Temporal Logic

B. Tofan, G. Schellhorn, S. Bäumler, W. Reif

Report 2010-07 August 2010

Institut für Informatik
D-86135 Augsburg

Embedding Rely-Guarantee Reasoning in Temporal Logic

Bogdan Tofan, Gerhard Schellhorn, Simon Bäumler, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg
D-86135 Augsburg, Germany
{tofan,schellhorn,baeumler,reif}@informatik.uni-augsburg.de

Abstract. The combination of temporal logic and rely-guarantee reasoning is a solid approach for the verification of concurrent programs. We describe a formalization of rely-guarantee reasoning within the temporal logic framework of the interactive prover KIV. Our previous encoding has been enhanced to permit simpler rely conditions and enriched to make it more expressive. Moreover, an instance of the new theory is defined to better exploit the symmetry inherent in many concurrent systems, by considering a single pair of processes only. We verify the resulting local proof obligations, applying symbolic execution to show memory safety, linearizability and lock-freedom of a shared stack that recycles memory.

Keywords: Verification, Temporal Logic, Compositional Reasoning, Rely-Guarantee, ABA Problem, Linearizability, Lock-Freedom

1 Introduction

Today's prevalent computers are multi-core systems that share a common memory space for inter-process communication. Much of recent research is focusing on the analysis of safety, correctness and liveness properties of concurrent programs for these architectures, trying to find suitable formal methods to deal with the challenges that arise from sharing common resources.

In 1976, Owicki and Gries described a fundamental technique for the formal analysis of shared memory concurrency [1]. Their Hoare-style proofs were based on predicate logic and code annotations at every point where interference could occur, requiring these properties to be preserved during the steps of other processes (or more generally, components). Since it is necessary to know the implementation details of concurrently running processes, their technique is not compositional.

In 1983, Jones' rely-guarantee method [2] (also known as assume/commit) essentially improved the Owicki-Gries approach, providing a compositional and scalable treatment of interference. Processes abstractly specify their expected environment behavior which permits to verify each process separately without relying on concrete implementation details of other processes.

Our approach is based on interval temporal logic [3] and applies symbolic execution (stepping forward through a process' code and calculating strongest post-conditions) for the compositional verification of concurrent programs. An embedding of rely-guarantee reasoning is used to reduce the proof effort that arises from non-deterministically interleaving process steps (previously described in [4]). The expressive framework permits to formalize and verify rely-guarantee reasoning and decomposition theorems to prove global safety, correctness and liveness properties of concurrent algorithms by looking at single processes only. This technical report focuses on improvements on this formalization, demonstrating its benefits by verifying a lock-free stack which reuses memory locations [5]. The main contributions are:

- The improved encoding of the rely-guarantee method provides a simple definition of the $\xrightarrow{+}$ operator [6] and simplifies rely conditions by removing the claim of stability of the invariant. Two further predicates are added to ensure the existence of suitable initial states and to describe conditions that hold in between executions of a process' concurrent operations (similar to the Hoare-style pre- and post conditions in [7]).
- An instance of this improved encoding is defined which better exploits the symmetry inherent in many concurrent systems. By splitting the overall program state into process-local and shared parts further simplifications for both specification and verification are achieved.
- Finally, we show the applicability of this instance by verifying memory safety (inlcuding absence of null dereferences and memory leaks), linearizability and lock-freedom of a (slightly optimized) shared stack which recycles memory. We have previously assumed garbage collection to avoid the intricacies of memory reuse in lock-free algorithms.

The report is structured as follows: Section 2 gives a brief introduction to KIV's temporal logic framework. Section 3 describes the improved encoding of rely-guarantee reasoning. Section 4 defines an instance of this theory which exploits the symmetry of many concurrent systems. Section 5 describes the stack and its verification based on this symmetric instance. We conclude with a section about related work and a summary (Section 6).

2 Temporal Logic in KIV

This section gives a short introduction to the temporal logic framework in KIV [8]. Further details can be found e.g. in [9].

2.1 Interval Temporal Logic

Our variant of interval temporal logic (ITL) [3] is based on predicate logic, algebras (to define a semantic for the signature) and intervals, i.e. finite or infinite sequences of states (a state maps variables to values in the algebra). In contrast

to standard ITL, our formalism explicitly includes the behavior of the program's environment in each step: in an interval $I = [I(0), I'(0), I(1), I'(1), \ldots]$ the first program transition leads from the initial state $I(0)$ to the primed state $I'(0)$ whereas the next transition (from state $I'(0)$ to $I(1)$) is a transition of the program's environment. In this manner program and environment transitions alternate (similar to [7]).

Variables are partitioned into static variables v (written lower case), which never change their value ($I(0)(v) = I'(0)(v) = I(1)(v) = \ldots$) and flexible variables V (starting with an uppercase letter) which can have different values in different states of an interval. We write V, V', V'' to denote variable V in states $I(0)$, $I'(0)$ and $I(1)$ respectively. In the last state (characterized by the atomic formula **last**) of an interval, the value of a primed or double primed variable is equal to the value of the unprimed variable by convention.

The logic uses standard connectives ($\wedge$, $\vee$, ...) as well as the usual temporal operators to express properties of an interval ($\Box$, $\Diamond$, $\bullet$, **until**, **unless**, ...). We usually write φ, ψ to indicate a formula (α, β to indicate a program). For instance, the standard semantics of an unless formula $\varphi\,\mathbf{unless}\,\psi$ requires φ to hold in every state of an interval or to hold until a state in which ψ holds is reached ($\Box\,\varphi \vee (\varphi\,\mathbf{until}\,\psi)$). In rely-guarantee proofs, formulas of the form $\varphi\,\mathbf{unless}\,(\varphi \wedge \neg\,\psi)$ are of particular interest. That is why an additional operator $\xrightarrow{+}$ ("sustains") is introduced to save the second writing of formula φ:

$$\psi \xrightarrow{+} \varphi \;:\equiv\; \varphi\,\mathbf{unless}\,(\varphi \wedge \neg\,\psi) \tag{1}$$

The programming language provides the common sequential constructs (:= , ; , **if**, ...), a construct for weak-fair ($\alpha \parallel \beta$) and one for non-fair ($\|_{\mathrm{nf}}$) interleaving. The usual (blocking) synchronization construct **await** φ is also supported: it blocks execution until the test φ is satisfied (not used in the non-blocking algorithms here).

Programs and formulas can be mixed arbitrarily since they both evaluate to true or false over an algebra $\mathcal{A}$ and an interval I (hence module implementations can be abstracted by temporal properties). A program evaluates to true ($\mathcal{A}, I \models \alpha$) if I is a possible run of the program.

2.2 Symbolic Execution and Induction

KIV is based on the sequent calculus. A sequent is an assertion of the form $\Gamma \vdash \Delta$, where Γ and Δ are lists of formulas, which means that the conjunction of all formulas in antecedent Γ implies the disjunction of all formulas in succedent Δ. Sequents are implicitly universally closed. A typical sequent (proof obligation) about interleaved programs has the form

$$\alpha, E, F \vdash \varphi$$

where an (interleaved) program α executes the system steps; the system's environment behavior is constrained by temporal formula E and F is a predicate

logic formula that describes the current state; φ is the property which has to be shown. For example, a sequent of the form mentioned above might be

$$(M := M + 1;\ \beta),\ \Box\ M'' \geq M',\ M = 1\ \vdash\ M'' = M' \xrightarrow{+} M' > M$$

The program executed is $(M := M + 1;\ \beta)$ where β is an arbitrary program and the environment is assumed never to decrement counter M (formula $\Box\ M'' \geq M'$). The current state maps M to 1 and it has to be shown that the program increments M if previous environment steps have not changed its value (formula $M'' = M' \xrightarrow{+} M' > M$).

Symbolic Execution. Proving sequents that contain temporal assertions is done by symbolically stepping forward to the next states of an interval. Executing such a step concerns both programs and formulas in two implicit phases.

In the first phase, information about the first step (both system and environment) is separated from information about the rest of the run. In the example above, from the assignment and the environment assumption we get $M' = M+1$ and $M'' \geq M'$ for the first step and $\bullet\ \beta$, respectively $\bullet\ \Box\ M'' > M'$ for the rest of the run. That is, executing a program is done by calculating the effect of the first statement and discarding it. An always formula is transcribed using the unwinding rule $\Box\ \varphi \equiv \varphi \wedge \bullet\ \Box\ \varphi$. Similarly, by applying the unwinding rule for the sustains operator

$$\psi \xrightarrow{+} \varphi\ \equiv\ \varphi \wedge (\psi \rightarrow \bullet\ (\psi \xrightarrow{+} \varphi))$$

the succedent is transcribed to $M' > M \wedge (M'' = M' \rightarrow \bullet\ (M'' = M' \xrightarrow{+} M' > M))$.

In the second phase of a symbolic execution step, the unprimed and primed variables M and M' are substituted with fresh static variables that describe the former state, whereas the double primed variable M'' is replaced with the unprimed variable M and leading next operators ($\bullet$) are dropped in the next state. Let variable M be replaced by m, respectively M' by m_0 in the example above. Then a symbolic execution step leads to a trivial predicate logic goal for the first system transition (essentially $m_0 = m + 1 \rightarrow m_0 > m$) and a sequent that describes the remaining interval from the next state on:

$$\beta,\ \Box\ M'' \geq M',\ M = 2\ \vdash\ M'' = M' \xrightarrow{+} M' > M$$

M has value 2 in the new state, because $M' > M$ has to be further sustained only if the environment leaves M unchanged.

Induction. Well-founded induction is used to deal with loops. For finite intervals it is possible to induce over the length of an interval. For infinite intervals a term for well-founded induction can often be derived from liveness properties $\Diamond\ \varphi$ as the number of steps N until φ holds:

$$\Diamond\ \varphi\ \leftrightarrow\ \exists\ N.\ (N = N'' + 1)\ \textbf{until}\ \varphi \tag{2}$$

The equivalence states that φ is eventually true, if and only if N can be decremented (note that $N = N'' + 1$ is equivalent to $N > 0 \wedge N'' = N - 1$) until φ

becomes true. Proving a formula of the form $\Box\ \varphi$ on infinite traces is then simply done by rewriting $\Box\ \varphi$ to $\neg \Diamond \neg\ \varphi$ and a proof by contradiction. Similarly, an induction term can be extracted from an **unless** formula using the equivalence

$$\varphi\ \textbf{unless}\ \psi\ \leftrightarrow\ \forall\ B.\ (\Diamond\ B) \rightarrow (\varphi\ \textbf{unless}\ (\varphi \wedge B \vee \psi)) \tag{3}$$

φ **unless** ψ is true if it is true on every prefix of the trace that is terminated by the first time when boolean variable B becomes true. This rewriting enables extraction of the liveness property $\Diamond\ B$ to prove that the initial unless formula holds, by applying well-founded induction over the number of steps until B is true (2). Based on (3) and the definition of the sustains operator (1), a term for well-founded induction can be extracted from sustains formulas using the equivalence

$$\psi \xrightarrow{+} \varphi\ \leftrightarrow\ \forall\ B.\ (\Diamond\ B) \rightarrow ((\psi \wedge \neg\ B) \xrightarrow{+} \varphi) \tag{4}$$

3 Rely-Guarantee Reasoning and Decomposition of Linearizability and Lock-Freedom

This section describes our concurrent system model and the improved encoding of rely-guarantee reasoning, outlining the benefits of the new formalization. Moreover, the decomposition theorems for linearizability and lock-freedom are briefly introduced (for details see [4] and [10]).

3.1 System Model

The parallel system CSpawn recursively spawns $n + 1$ processes ($n : nat$) to execute a generic operation CSeq in parallel (**if*** denotes that evaluating the if-condition requires no extra step).

```
CSpawn(n; Act, In, CS, Out) {
 if* n = 0 then
  CSeq(n; Act, In, CS, Out)
 else
    CSeq(n; Act, In, CS, Out)
  || CSpawn(n − 1; Act, In, CS, Out)}
```

```
CSeq(p; Act, In, CS, Out) {
 {   skip
   ∨ {Act(p) := true;
      COP(p, In; CS, Out);
      Act(p) := false}
 }*}
```

Operation CSeq finitely or infinitely often (denoted by *) non-deterministically chooses ($\vee$) between two possible behaviors: either it executes a no operation *skip* (this models computations that are not directly related to the investigated algorithm) or it executes one of the operations of the underlying algorithm (modeled as COP). In the stack case study for instance (cf. Section 5), COP is the nondeterministic choice between one of the two concrete operations push and pop.

CSeq's value parameter $p : nat$ represents the identifier of the invoking process. Reference parameter $Act : nat \rightarrow bool$ is a boolean function which is

used to distinguish whether a process is currently executing COP (mainly used to lift liveness properties). Function $In : nat \to input$ passes an arbitrary input value $In(p)$ to COP. Since In is a reference parameter in CSeq, whenever COP is invoked its input value can differ from previous invocations. The remaining parameters include a generic state variable $CS : cstate$ and an output function $Out : nat \to output$ to return values. All processes work on the overall program state CS which abstracts all local states and the global state.

3.2 Embedding Rely-Guarantee Reasoning in ITL

Reasoning about properties of the (interleaved) executions of CSpawn is tedious work. The goal of rely-guarantee reasoning is to avoid such global reasoning and examine executions of single processes instead, trying to find suitable process-local properties (of COP, resp. CSeq) which can be lifted to global ones. To achieve such a decomposition, the behavior of other processes is abstracted using a rely predicate $R_p : cstate \times cstate$ which describes p's expected environment behavior. In return, each process guarantees a certain behavior towards its environment using a guarantee predicate $G_p : cstate \times cstate$. The first parameter of a rely/guarantee condition corresponds to the state before the environment/system transition and the second argument denotes the next state.

An important relation between guarantees and relies is that each guarantee must preserve the rely conditions of all other processes.

$$q \neq p \wedge G_p(CS_0, CS_1) \to R_q(CS_0, CS_1) \tag{5}$$

The main local temporal property requires each process to sustain its guarantee if preceding environment steps have not violated its rely condition.

$$\mathrm{COP}(p, In; CS, Out) \vdash R_p(CS', CS'') \xrightarrow{+} G_p(CS, CS')$$

In order to propagate this property to executions of CSeq, guarantee conditions must be reflexive (in case of skip or a step that sets the activity flag, the current state stays the same). The lifting of this local property to the level of interleaved executions of CSpawn, requires the ability to summarize several consecutive rely steps, i.e. we claim R_p to be transitive.

$$\begin{array}{l} G_p(CS, CS) \\ R_p(CS_0, CS_1) \wedge R_p(CS_1, CS_2) \to R_p(CS_0, CS_2) \end{array} \tag{6}$$

Reasoning in practice strongly relies on suitable invariant properties of the analyzed algorithm. This is why our rely-guarantee encoding also provides an invariant predicate $Inv : cstate$. To ensure that this predicate holds in every state, it is sufficient to claim its stability over rely steps, i.e. $Inv(CS_0) \wedge R_p(CS_0, CS_1) \to Inv(CS_1)$. The disadvantage of this requirement is that the rely condition (and the guarantee) becomes unnecessarily strong. Alternatively one could require stability over the guarantee condition only, but we prefer to completely decouple both rely and guarantee conditions from the invariant and to claim

its preservation (both by system and environment transitions) separately, as a further property to be sustained.

$$\begin{array}{ll} & \mathrm{COP}(p, In; CS, Out),\ Inv(CS) \\ \vdash & \quad (R_p(CS', CS'') \wedge (Inv(CS') \rightarrow Inv(CS''))) \\ & \xrightarrow{+} (G_p(CS, CS') \wedge (Inv(CS) \rightarrow Inv(CS'))) \end{array}$$

Properties which hold in between executions of COP only (similar to the pre- and post conditions in [7]) can not be encoded within the invariant since this predicate must hold in intermediate states of COP as well. Therefore a further predicate $Idle_p : cstate$ is defined which is unchanged by other processes and (re)established in the last state of each run.

$$\begin{array}{ll} & \mathrm{COP}(p, In; CS, Out),\ Inv(CS),\ Idle_p(CS) \\ \vdash & \quad (\quad R_p(CS', CS'') \wedge (Inv(CS') \rightarrow Inv(CS'')) \\ & \qquad \wedge (Idle_p(CS'') \leftrightarrow Idle_p(CS'))) \\ & \xrightarrow{+} (\quad G_p(CS, CS') \wedge (Inv(CS) \rightarrow Inv(CS')) \\ & \qquad \wedge (\mathbf{last} \rightarrow Idle_p(CS)) \\ & \qquad \wedge (\forall\ q \neq p.\ (Idle_q(CS') \leftrightarrow Idle_q(CS)))) \end{array} \tag{7}$$

To prove the existence of suitable initial states which establish the idle and invariant properties, a further predicate $Init : cstate$ is used. It must ensure that initially all processes are idle and the invariant holds.

$$\begin{array}{l} (\exists\ CS.\ Init(CS)) \\ Init(CS) \rightarrow Idle_p(CS) \wedge Inv(CS) \end{array} \tag{8}$$

Since the generic setting also takes into account the environment of the overall system, a global rely condition R_S is required too. It preserves all local rely conditions, the invariant, all idle predicates and prohibits changes to the activity function by the system's environment. The rely-guarantee formulas (5) to (8) plus the restrictions on the global environment are sufficient to establish the following three properties to hold in every interleaved execution: each step preserves its local guarantee, the invariant holds in every state and whenever a process is not active, it is an idle state.

Theorem 1 (Rely-Guarantee).
If formulas (5) to (8) can be proved (for some $R_S, R_p, G_p, Idle_p, Init, Inv$), then:

$$\begin{array}{ll} & \textsc{Cspawn}(n; \ldots),\ \Box\ R_S,\ Init(CS),\ \forall\ p \leq n.\ \neg\ Act(p) \\ \vdash \Box\ (& \quad (\exists\ p \leq n.\ G_p(CS, CS')) \wedge Inv(CS) \wedge Inv(CS') \\ & \qquad \wedge (\forall\ p \leq n.\ \neg\ Act(p) \rightarrow Idle_p(CS))) \end{array}$$

The proof for the improved theorem proceeds much like the old one [4]. It is important to notice that Theorem 1 has a major significance for verification in practice. As the identified rely-guarantee and invariant conditions hold at all times in each concurrent execution they can be used to establish other properties of interest, here memory safety, linearizability and lock-freedom.

3.3 Decomposition of Linearizability and Lock-Freedom

Linearizability [11] is a global correctness property for concurrent algorithms (similar to serializability). It claims for each interleaved execution the existence of an equivalent and legal sequential execution which preserves the order of non-concurrent runs. This global property can be broken down to a local proof obligation which requires operations to appear to take effect instantly at one point (the linearization point) between their invocation and return. Using Theorem 1 the refinement-based proof of linearizability requires each concrete run of COP to refine an abstract run of operation AOP (concrete states are mapped to abstract states $AS : astate$ using the representation predicate $Abs : astate \times cstate$). The abstract executions consist of an atomic step (the linearization point) which is preceded and followed by skip steps.

Theorem 2 (Linearizability).
CSpawn *is linearizable if formulas (5) to (8) hold and:*

$$\begin{array}{l} \quad \mathrm{COP}(p, In; CS, Out), \\ \quad \Box\ (R_p(CS', CS'') \wedge Inv(CS) \wedge Inv(CS')),\ Idle(CS) \\ \vdash \exists\ AS.\ \mathrm{AOP}(p, In; AS, Out) \wedge \Box\ (Abs(AS, CS) \wedge Abs(AS', CS')) \end{array}$$

Lock-freedom is a global liveness property which requires that at all times in a concurrent execution, one of the running operations eventually completes [12]. Our decomposition technique reduces the proof of lock-freedom to a local proof obligation based on finding a suitable predicate $U : cstate \times cstate$ which describes interference freedom. The local proof obligation requires showing that each local execution terminates if it updates the state (corresponding to the linearization point), or if it encounters no interference.

Theorem 3 (Lock-Freedom).
CSpawn *is lock-free if formulas (5) to (8) hold and there is a reflexive and transitive predicate U with:*

$$\begin{array}{l} \quad \mathrm{COP}(p, In; CS, Out), \\ \quad \Box\ (R_p(CS', CS'') \wedge Inv(CS) \wedge Inv(CS')),\ Idle(CS) \\ \vdash \Box\ (\neg\ U(CS, CS') \vee (\Box\ U(CS', CS'')) \rightarrow \Diamond\ \mathbf{last}) \end{array}$$

4 Exploiting Symmetry

The generic setting presented in Sections 3.1 and 3.2 provides process-specific operations, relies and guarantees. However, the verification of many concurrent algorithms which are inherently symmetric (all processes execute the same code) does not require this full generality. The symmetry can be exploited to break down the analysis of the overall system into analyzing just two parallel processes: a process p and one other process q (abstracted by p's rely condition) which represents all other processes. Since the previous theory does not have this localized view of two processes, we refer to it as the "global theory" and call its symmetric instance the "local theory" in the following.

4.1 Symmetric Rely-Guarantee Instance

Other processes are abstracted by p's rely condition in the global theory, but all processes work on the same generic program state CS (including all local states and the global state). Therefore, properties typically must be specified globally, i.e. based on variable functions (for each process and local variable) and universal quantification over all process identifiers. This can lead to a lower degree of proof automation in practice as a suitable quantifier instantiation can not always be deduced automatically.

These shortcomings are avoided by introducing a localized view of the (symmetric) system. The key idea is to split the overall state CS into its process-local and global parts by instantiating it with a pair $LSF \times GS$ consisting of a local state function $LSF : nat \to lstate$ (mapping each process to its local state) and a global state variable $GS : gstate$. This allows for explicitly denoting individual local states $LS, LSQ : lstate$ where LS represents the local state of process p ($LSF(p)$) and LSQ the local state of process q ($LSF(q)$).

Since the notion of a heap is not an inherent part of our logic (in contrast to separation logic, the heap is a part of the global state), it is occasionally necessary to explicitly encode disjointness properties of local states in practice. That is why a symmetric predicate $LDisj : lstate \times lstate$ is added to the theory (in the global theory, disjointness properties are part of the invariant and usually quantified over all pairs of process identifiers).

$$LDisj(LS, LSQ) \to LDisj(LSQ, LS)$$

Similar to the global theory, it is required that initial states establish the (local) idle predicate, invariant and disjointness. The existence of such initial states has to be shown for all local states. This is the only place where the new theory still has a global flavor though.

$$\exists\, LSF, GS.\ GInit(GS) \wedge \forall\, p.\ LInit(LSF(p))$$
$$LInit(LS) \to LIdle(LS)$$
$$LInit(LS) \wedge GInit(GS) \to LInv(LS, GS)$$
$$LInit(LS) \wedge LInit(LSQ) \to LDisj(LS, LSQ)$$

The local guarantee $LG : lstate \times lstate \times gstate \times lstate \times gstate$ still has to be reflexive. Its first three parameters represent the two local states and the global state before a system transition (the first parameter is the local state of the running process). The last two parameters are the local state of the running process and the global state after a system transition. The local state of the other process (second parameter) implicitly remains unchanged in a step. The localized guarantee still has to preserve the transitive rely $LR : lstate \times gstate \times gstate$ of the other process. Its parameters correspond to the local state of one of both processes (depending on the context) and the global state before and after a transition.

$$LG(LS, LSQ, GS, LS, GS)$$
$$LR(LS, GS_0, GS_1) \wedge LR(LS, GS_1, GS_2) \to LR(LS, GS_0, GS_2)$$
$$LG(LS_0, LSQ_0, GS_0, LS_1, GS_1) \to LR(LSQ_0, GS_0, GS_1)$$

The main local temporal property (corresponding to (7)), rephrased in terms of a single pair of processes, claims that during the execution of a localized procedure LCOP (starting in an idle state in which the invariant and disjointness holds for both processes), an extended (localized) guarantee (syntactically abbreviated as LG_{ext}) is sustained if preceding environment steps have preserved an extended (localized) rely LR_{ext}.

$$\begin{array}{l} \mathrm{LCOP}(I; LS, GS, O),\ LIdle(LS), \\ LInv(LS, GS),\ LInv(LSQ, GS),\ LDisj(LS, LSQ) \\ \vdash LR_{ext} \xrightarrow{+} LG_{ext} \end{array} \tag{9}$$

Operation LCOP is called with input and output variables $I : input$, $O : output$ (in the global theory these are functions), the local state of the running process and the global state. The extended guarantee requires the local guarantee (hence the rely of the other process) to be preserved, the idle predicate to hold in each final state and the invariant (for both processes) and disjointness to be preserved.

$$\begin{array}{rcl} LG_{ext} & :\equiv & LG(LS, LSQ, GS, LS', GS') \wedge (\mathbf{last} \rightarrow LIdle(LS)) \\ & & \wedge\ (\quad LInv(LS, GS) \wedge LInv(LSQ, GS) \wedge LDisj(LS, LSQ) \\ & & \quad \rightarrow LInv(LS', GS') \wedge LInv(LSQ', GS') \wedge LDisj(LS', LSQ')) \end{array}$$

The extended rely expects the environment not to change the local state of the executing process (this has to be coded within the rely in the global theory) and to maintain the local rely, the invariant and disjointness.

$$\begin{array}{rcl} LR_{ext} & :\equiv & LS'' = LS' \wedge LR(LS', GS', GS'') \\ & & \wedge\ (LInv(LS', GS') \rightarrow LInv(LS'', GS'')) \\ & & \wedge\ (LInv(LSQ', GS') \rightarrow LInv(LSQ'', GS'')) \\ & & \wedge\ (LDisj(LS', LSQ') \rightarrow LDisj(LS'', LSQ'')) \end{array}$$

Process identifiers are no explicit part of the local theory. They are not required in the case study (see Section 5) and if they were needed, they could be simply defined as part of the local state. Furthermore, there is no upper bound on process identifiers which are just natural numbers. If there is a need for an upper bound in applications, it is possible to introduce a constant bound, letting processes with a higher identifier just execute a skip step.

To prove that the new theory is indeed an instance of the global theory, the global proof obligations (5) to (8) must follow from the local ones based on suitable instantiations of the global signature. The initial restrictions from the global theory are instantiated with the corresponding restrictions on all local states and the global state. The global idle predicate is equivalent to the corresponding local predicate.

$$\begin{array}{l} Init(LSF, GS) \leftrightarrow GInit(GS) \wedge \forall\ p.\ LInit(LSF(p)) \\ Idle_p(LSF, GS) \leftrightarrow LIdle(LSF(p)) \end{array}$$

The global invariant corresponds to all local invariants plus the disjointness predicate for all pairs of local states.

$$\begin{array}{rcl} Inv(LSF, GS) \leftrightarrow & & (\forall\ p.\ LInv(LSF(p), GS)) \\ & & \wedge\ (\forall\ p \neq q.\ LDisj(LSF(p), LSF(q))) \end{array}$$

The global rely condition is mapped to the local rely and the assumption that the local state is not changed by the environment. Finally, the global guarantee simply preserves all global relies.

$$\begin{aligned} R_p(LSF', GS', LSF'', GS'') \leftrightarrow \quad & LSF''(p) = LSF'(p) \\ & \wedge\, LR(LSF'(p), GS', GS'') \\ G_p(LSF_0, GS_0, LSF_1, GS_1) \leftrightarrow \forall\, q \neq p.\ & R_q(LSF_0, GS_0, LSF_1, GS_1) \end{aligned}$$

4.2 Local Proof Obligations for Linearizability and Lock-Freedom

Within this symmetric instance, the proof obligations for linearizability and lock-freedom are fully local, i.e. they only take into account the local state of the executing process and the global state.

$$\begin{aligned} & \mathrm{LCOP}(I; LS, GS, O),\ LIdle(LS), \\ & \Box\,(\quad LS'' = LS' \wedge LR(LS', GS', GS'') \\ & \qquad \wedge\, LInv(LS, GS) \wedge LInv(LS', GS')) \\ \vdash\ & \exists\, AS.\ \mathrm{LAOP}(I; AS, O) \wedge \Box\,(LAbs(AS, GS) \wedge LAbs(AS', GS')) \end{aligned} \tag{10}$$

The new (localized) unchanged predicate for lock-freedom is defined purely on global state transitions $LU : gstate \times gstate$.

$$\begin{aligned} & \mathrm{LCOP}(I; LS, GS, O),\ LIdle(LS), \\ & \Box\,(\quad LS'' = LS' \wedge LR(LS', GS', GS'') \\ & \qquad \wedge\, LInv(LS, GS) \wedge LInv(LS', GS')) \\ \vdash\ & \Box\,(\neg\, LU(GS, GS') \vee (\Box\, LU(GS', GS'')) \rightarrow \Diamond\, \mathbf{last}) \end{aligned} \tag{11}$$

The global counterparts for the (local) representation and unchanged predicates are easy to find. If verification of (10) or (11) in case studies requires more information, the always formula in the antecedent can be enriched by the invariant for the other process or disjointness properties (not required here).

5 A Case Study: Treiber's Stack

Lock-free concurrent algorithms try to avoid the problems of conventional lock-based implementations (e.g. deadlocks, priority inversion) by applying an optimistic try-retry scheme and atomic synchronization primitives such as CAS (Compare And Swap) instead of locks.

$$\begin{aligned} & \mathrm{CAS}(Old, New; G, Succ)\ \{ \\ & \ \mathbf{if^*}\ G = Old\ \mathbf{then}\ \{G := New,\ Succ := true\}\ \mathbf{else}\ \{Succ := false\}\} \end{aligned}$$

CAS compares a global value G with a local value *Old* (an older snapshot version of G). If these values are equal, then G is updated to a new value *New* and boolean flag *Succ* is set to true to indicate a successful update. Otherwise the flag is set to false indicating that no update has occurred. Since CAS executes atomically (a comma separates parallel assignments), evaluating the if-condition requires no extra step.

5.1 The Stack Algorithm

CAS does not guarantee that the current value A of G has not been changed since it was stored in *Old*. In the meantime, other processes can change G to B and then back to A. Such undetected intermediate modifications can cause problems like data structure corruption or wrong return values (ABA problem). A simple solution is attaching a counter to shared locations and increasing it with every modification (this solution requires atomic double-width read and CAS instructions; the possibility not to detect changes due to a wrap around of the counter can be disregarded in practice [5]).

Treiber's stack algorithm (see Figure 1) initiates a modification counter to detect changes to the shared variable *Top* (pairs of references and natural numbers with constructor rc and selector functions .ref and .cnt) which marks the top cell (pairs of values and references along with .val and .nxt selector functions) of the stack. The global data structure is represented in the application's memory *Hp* as a singly linked list of cells (area **Stack** in the figure). In programming environments without garbage collection, cells that are removed from the stack should be deallocated to avoid memory leaks. However, simple deallocation can lead to memory access faults. To still reduce memory usage, cells which are removed from the stack are inserted into a global set of pointers *Free* (heap area **Free** in the figure).

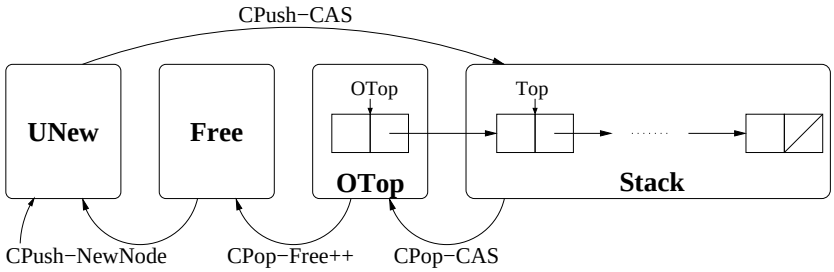
Whenever a process executes a push it tries to reuse a node from the set of free references instead of constantly allocating new references (cf. *NewNode* in Figure 1). Then it stores the shared top (both pointer and counter) in a local snapshot and sets the new node's next reference to the snapshot pointer. Using CAS the top pointer is updated and its counter is incremented if both values are equal to the snapshot (otherwise the loop is reiterated).

Similarly, a pop operation takes a snapshot and (if the associated pointer is not null) locally stores its next reference. This local next reference is the target of the subsequent CAS (without the increment of the modification counter in push, this instruction might succeed when it should not, due to memory reuse, corrupting the stack). If it succeeds, the cell is removed from the stack, added to the free-set and its value is returned (otherwise the loop is reiterated).

5.2 Verifying the Stack.

In this section we describe the instantiation of the local rely-guarantee theory. The identified relies, guarantees and invariant properties are sufficient to prove the absence of access errors and memory leaks, linearizability and lock-freedom.

Rely-Guarantee Conditions and Invariant. The generic LCOP operation is instantiated with the nondeterministic choice between one of the two stack operations. The generic local state variable *LS* becomes a tuple consisting of variables *UNew*, *USucc*, *OTop*, *OSucc* (for the other local state, variables *UNewq*, *USuccq*, *OTopq*, *OSuccq* are used). The global state *GS* corresponds to *Top*, *Hp* and *Free*.




```
CPush(I; UNew, USucc, Top, Hp, Free) {
 NewNode(I; UNew, USucc, Hp, Free);
  let UTop = [?] in {
   while ¬ USucc do {
    UTop := Top;
    Hp[UNew].nxt := UTop.ref;
    CAS(UTop, rc(UNew, UTop.cnt + 1);
          Top, USucc) }}}

NewNode(I; UNew, USucc, Hp, Free) {
 choose Ref with (Ref ≠ null
  ∧ (Free = ∅ → ¬ Ref ∈ Hp)
  ∧ (Free ≠ ∅ → Ref ∈ Free)) in {
  Hp := Hp ∪ {Ref}, Free := Free \ {Ref},
  Hp[Ref].val := I, USucc := false,
  UNew := Ref }}
```

```
O1  CPop(; OTop, OSucc, Top, Hp, Free, O) {
O2   let Lo = empty, ONxt = [?] in {
O3    while ¬ OSucc do {
O4     OTop := Top;
O5     if OTop.ref = null then {
O6      OSucc := true;
O7     } else {
O8      ONxt := Hp[OTop.ref].nxt;
O9      CAS(OTop, rc(ONxt, OTop.cnt);
O10           Top, OSucc) }}
O11   if OTop.ref ≠ null then {
O12    Lo := Hp[OTop.ref].val′;
O13    Free := Free ∪ {OTop.ref},
O14    OSucc := false }
O15   O := Lo, OSucc := false }}
```

Fig. 1. Declaration of push and pop in KIV.

In idle states the loop flag for push is true and the flag for pop is false (these flags are also used to mark code areas since the logic does not provide program counters). Moreover, in initial states the local new and top pointers ("local top" relates to the snapshot pointer in pop in the following) are null and the stack, the free-set and heap are empty.

The invariant ensures a valid heap pointer structure, i.e. the stack represents a finite list. The local new and top pointers are either null or allocated. Furthermore, they are disjoint from the stack: new cells are not in the stack before being successfully pushed (i.e. when $\neg$ *USucc* holds; see heap area **UNew** in Figure 1) and local top pointers are not part of the stack after being popped as long as they are not freed (i.e. when $OTop.\mathrm{ref} \neq null \wedge OSucc$ holds; see area **OTop** in the figure). Moreover, all references within the free-set are not null, allocated and disjoint from the stack, the local (not yet pushed) new variables, and the (removed but not freed) top variables. The invariant implies that null dereferences do not occur during execution.

Disjointness of both local states relates to new cells (not yet pushed) and local top references (removed but not yet free): these cells are disjoint among

each other (i.e. $UNew \neq UNewq$, $OTop \neq OTopq$) and new cells are disjoint from the other process' top and vice versa.

For proving correctness of the stack implementation (linearizability), three main rely conditions have to be established. First, the content of new cells remains unchanged by the environment during the push-loop.

$$UNew' \neq null \wedge \neg\ USucc' \rightarrow Hp''[UNew''] = Hp'[UNew']$$

Second, during the pop-loop, the global top and the local top's next pointer remain unchanged, or if the local top pointer becomes reachable (again) the counter has been increased (due to the increment in push).

$$\begin{array}{ll} & \neg\ OSucc' \wedge OTop'.\text{ref} \neq null \wedge OTop' = Top' \\ \rightarrow & \quad Top'' = Top' \wedge Hp''[OTop''.\text{ref}].\text{nxt} = Hp'[OTop'.\text{ref}].\text{nxt} \\ & \vee \neg\ reachable(Top''.\text{ref}, OTop''.\text{ref}, Hp'') \\ & \vee\ Top''.\text{cnt} > Top'.\text{cnt} \end{array} \tag{12}$$

Third, after a cell has been removed from the stack but not yet freed, its value is unchanged and the cell remains disjoint from the stack.

$$\begin{array}{ll} & OSucc' \wedge OTop'.\text{ref} \neq null \\ & \wedge \neg\ reachable(Top'.\text{ref}, OTop'.\text{ref}, Hp') \\ \rightarrow & \quad Hp''[OTop''.\text{ref}].\text{val} = Hp'[OTop'.\text{ref}].\text{val} \\ & \wedge \neg\ reachable(Top''.\text{ref}, OTop''.\text{ref}, Hp'') \end{array} \tag{13}$$

Two further rely properties cause the pop-loop to retry if the local top is removed from the stack before the CAS: the modification counter is never decremented and whenever the snapshot is inserted in the stack, the modification counter grows.

$$\begin{array}{ll} & Top''.\text{cnt} \geq Top'.\text{cnt} \\ \wedge\ (& \quad \neg\ OSucc' \wedge OTop'.\text{ref} \neq null \\ & \wedge \neg\ reachable(Top'.\text{ref}, OTop'.\text{ref}, Hp') \\ & \rightarrow (reachable(Top''.\text{ref}, OTop''.\text{ref}, Hp'') \rightarrow Top''.\text{cnt} > Top'.\text{cnt})) \end{array} \tag{14}$$

The guarantee is weakly defined to maintain the rely condition of the other process. Furthermore, it ensures that each step preserves ownership of previously owned references which are not inserted in the stack or free-set. A reference *Ref* is owned if it is a local new or top reference.

$$\begin{array}{ll} & Owns(Ref, LS) \\ \leftrightarrow & \quad (\neg\ USucc \wedge UNew = Ref) \\ & \vee\ (OSucc \wedge OTop.\text{ref} \neq null \wedge OTop.\text{ref} = Ref) \end{array}$$

Each local step preserves (or establishes) ownership of cells which are allocated (or reused), removed from the stack or previously owned and not inserted in the stack or the free-set.

$$\begin{array}{l} PreservesNoLeak(LS, GS, LS', GS') \leftrightarrow \forall\ Ref \in Hp'. \\ \quad (\quad \neg\ Ref \in Hp \vee Ref \in Free \\ \quad \vee\ reachable(Top.\text{ref}, OTop.\text{ref}, Hp) \vee Owns(Ref, LS)) \\ \rightarrow (\quad reachable(Top'.\text{ref}, OTop'.\text{ref}, Hp') \\ \quad \vee\ Ref \in Free' \vee Owns(Ref, LS')) \end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots \quad \dots \quad \dots}{\mathbf{O6},\ S, ((=) \vee (-) \vee (+)) \vdash \dots}\ (4)}{\mathbf{O5},\ S, \mathit{Top} \neq \mathit{null}, (=) \vdash \dots}\ (3) \qquad \mathbf{O5},\ S, (-) \vdash \dots \qquad \mathbf{O5},\ S, (+) \vdash \dots}{\mathbf{O5},\ S, ((=) \vee (-) \vee (+)) \vdash \dots}\ (2)}{\mathbf{O4},\ S, \mathit{Top} \neq \mathit{null} \vdash (R_{ext} \wedge \neg\, B) \xrightarrow{+} G_{ext}}\ (1)$$

$$\begin{aligned} S :\equiv\ & ((N = N'' + 1)\, \mathbf{until}\, B) \wedge \mathit{IH} \wedge \neg\, \mathit{OSucc} \wedge \mathit{Lo} = \mathit{empty} \\ & \wedge\, \mathit{LInv}(\mathit{LS}, \mathit{GS}) \wedge \mathit{LInv}(\mathit{LSQ}, \mathit{GS}) \wedge \mathit{LDisj}(\mathit{LS}, \mathit{LSQ}) \\ (=) :\equiv\ & \mathit{Top} = \mathit{OTop}; \quad (-)\ :\equiv\ \neg\, \mathit{reachable}(\mathit{Top}, \mathit{OTop}, \mathit{Hp}) \\ (+) :\equiv\ & \mathit{Top}.\mathrm{cnt} > \mathit{OTop}.\mathrm{cnt} \end{aligned}$$

Fig. 2. Proof outline for pop.

This property can be lifted to prove that memory leaks do not occur. Starting in an initial state without leaks and knowing that each system step preserves its local guarantee at all times, heap locations are always either in the stack, the free-set or in a local new/top variable.

Proof Outline. Obviously, an initial state exists which is idle and establishes the invariant and symmetric disjointness properties. It is also simple to show that the guarantee is reflexive and implies the transitive rely condition.

Figure 2 outlines the (remaining) proof of temporal property (9) to demonstrate how the extended guarantee is maintained during the symbolic execution of LCOP. The focus is on the more challenging pop operation only.

Before symbolically executing the first step to enter the loop, the induction term N for well-founded induction is extracted from the $\xrightarrow{+}$ formula. The resulting induction hypothesis *IH* can be applied whenever a similar configuration is reached again and N has been decremented. The first symbolic execution step (a skip for evaluating the loop condition, followed by an environment transition) preserves the extended guarantee. If the last environment transition has preserved LR_{ext}, then the invariant and disjointness properties hold again in the new state (formula S) and the remaining program is the loop body starting from line 4, followed again by the loop (denoted as **O4**).

The proof in case of a non-empty stack ($\mathit{Top} \neq \mathit{null}$ in the root of the proof-tree) proceeds as follows. Proof step (1) is a symbolic execution step which locally stores the shared top, followed by an environment transition. This step preserves the extended guarantee and the proof continues knowing that the last environment transition has preserved the extended rely. In the next state the remaining program is **O5** and according to rely condition (12) there are three possible cases (depending on environment behavior). Proof step (2) discerns these cases. In the third premise (counting from left to right), the modification counter has been incremented (+) and in the second premise the local top is not

in the stack anymore (–). In both cases the rely conditions (14) ensure that the loop must be reiterated and the evolving sequents are closed inductively. In the first premise, the global top has not been modified, i.e. the current iteration can still succeed (=).

The proof applies the same pattern again in proof steps (3) and (4) (symbolic execution and a case distinction to discern environment behavior) until CAS is executed. In the leftmost branch of the proof-tree (corresponding to a successful loop), the next reference of the local top is not modified by the environment according to the first disjunct of (12), i.e. the CAS instruction reflects a correct pop and maintains a valid stack. In the remaining steps the removed cell is owned by the running process and its value remains unchanged (13), i.e. pop indeed returns the value that has been popped from the stack (needed for the refinement-based proof of linearizability).

Based on the established rely and invariant conditions, proving linearizability (10) is analogous to [4]. Proving lock-freedom (11) is much simpler for the stack then it was for the queue in [10] (the "unchanged" predicate which ensures termination of LCOP simply claims the shared top to remain unchanged by the environment).

6 Related Work and Summary

6.1 Related Work

A lot of interesting research has been done both on the rely-guarantee method and the verification of lock-free algorithms which we can not address here, limiting our considerations to some related approaches. To the best of our knowledge, no other approach is currently applied to mechanically verify memory safety (inlcuding absence of null dereferences and memory leaks), linearizability and lock-freedom of non-trivial lock-free algorithms using one logical framework only.

Nieto et al. [13] present the formalization of rely-guarantee reasoning in the interactive theorem prover Isabelle. Their compositional approach is based on an encoding into higher- order logic (instead of temporal logic). This has the advantage that soundness can be reduced to the soundness of higher-order logic, but the semantic encoding introduces a relatively large overhead. Their Hoare-style approach does not exploit symmetry and specifies relies and guarantees over the entire program state. We are not aware of an application of this encoding to verify fine-grained concurrency.

Pasareanu et al. [14] present a compositional framework for (symmetric and asymmetric) rely-guarantee based verification. Rely conditions are incrementally constructed by automatically learning from counter examples that are found by locally model checking single components. As far as we know, the approach concentrates on safety properties and has not been applied to verify lock-free algorithms.

Yahav et al. [15] describe their experience in using SPIN to model check linearizability of non-blocking concurrent algorithms. The considered algorithms

include a set implementation [16] with linearization points outside of the code of an executing operation. They assume garbage collection to avoid complications resulting from manual memory management and try to apply symmetry reduction to optimize the applicability of their approach when all processes execute the same operation. Their experimental results are yet limited to small system sizes.

A fully automatic approach for verifying linearizability is taken by Vafeiadis et al. [17] based on separation logic [18] and rely-guarantee reasoning. RGSep splits the program state in local and global parts and profits from the implicit treatment of disjoint heap locations by the separating conjunction operator which allows for specifying their relies and guarantees over the shared state only. The logic is restricted to safety analysis and the derivation of their proof obligations is not mechanized. They verify linearizability of a simplified version of the stack which is stripped off memory management code.

A rather different approach is taken by Reps et al. [19] based on abstract interpretation to automatically prove linearizability of several lock-free data structures. Their experimental results (in TVLA) are yet reduced to small system sizes and their verification relies on garbage collection and does not explicitly consider memory reuse.

Groves et al. [20] present a pen-and-paper proof of linearizability of a non-trivial elimination stack algorithm [21] based on trace-reduction and incremental refinement. Furthermore, they were the first to prove linearizability of a non-trivial queue algorithm (including modification counters) [22]. Their proof is automata-based and mechanized in PVS. In contrast to their monolithic proof, our proof for the queue does not require to encode programs as automata using program counters. Another difference is that their technique needs an intermediate automaton and backward simulation whereas our approach does not require any additional techniques.

6.2 Summary

We have described an improved generic embedding of rely-guarantee reasoning in the temporal logic framework of KIV. Moreover, we have derived a more symmetric instance of this theory and showed its application on a lock-free stack.

Currently, we are investigating the verification of a refined stack algorithm which replaces the algebraically specified free-set by a Treiber-like stack. We have successfully applied a similar symmetric instance (comprising the other process' local state in the rely condition) on the stack, extended by the hazard pointer algorithm from [23] which ensures safe memory reclamation. Improvements on the theory for proving linearizability (including a formal definition of linearizability [24] and the treatment of external linearization points), lock-freedom (integrating a more adequate notion of failure) and further case studies are part of future work. We are also interested in the integration of automatic techniques (shape analysis) within the prover to achieve a higher degree of automation.

References

1. Owicki, S.S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Inf. **6** (1976) 319–340
2. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, North-Holland (1983) 321–332
3. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge Univ. Press (1986)
4. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. Formal Aspects of Computing (FAC) (2009)
5. Treiber, R.K.: System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
6. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems (1995)
7. DeRoever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
8. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In Bibel, W., Schmitt, P., eds.: Automated Deduction—A Basis for Applications. Volume II: Systems and Implementation Techniques. Kluwer Academic Publishers, Dordrecht (1998) 13 – 39
9. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications **23**((2,3)) (2010) 285–307
10. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal logic verification of lock-freedom. In: Proc. MPC 2010. Springer LNCS (2010)
11. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3) (1990) 463–492
12. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. SIGOPS Oper. Syst. Rev. **26**(2) (1992) 108
13. Prensa Nieto, L.: The rely-guarantee method in Isabelle /HOL. In Degano, P., ed.: ESOP'03. Volume 2618 of LNCS., Springer (2003) 348–362
14. C. Pasareanu, D. Giannakopoulou, M.B.J.C.H.B.: Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design (2008)
15. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software, Springer-Verlag (2009) 261–278
16. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. Parallel Processing Letters **17**(4) (2007) 411–424
17. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
18. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. Lecture Notes in Computer Science **2142** (2001)
19. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: CAV. (2007) 477–490

20. Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. Formal Aspects of Computing (FAC) **21**(1–2) (2009) 187–223
21. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA '04: ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM Press (2004) 206–215
22. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE 2004. Volume 3235 of LNCS. (2004) 97–114
23. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6) (2004) 491–504
24. Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizabilty via non-atomic refinement. In J. Davies, J.G., ed.: Proceedings of the International Conference on integrated formal methods (iFM) 2007. Volume 4591 of LNCS., Springer (2007) 195–214